CHAPTER 7: INPUT AND OUTPUT

Input and output facilities are not part of the C language, so we have de-emphasized them in our presentation thus far. Nonetheless, real programs do interact with their environment in much more complicated ways than those we have shown before. In this chapter we will describe "the standard I/O library," a set of functions designed to provide a standard I/O system for C programs. The functions are intended to present a convenient programming interface, yet reflect only operations that can be provided on most modern operating systems. The routines are efficient enough that users will not feel the need to circumvent them "for efficiency" regardless of how critical the application. Finally, the routines are meant to be "portable," in the sense that they will exist in compatible form on any system where C exists, and that programs which confine their system interactions to facilities provided by the standard library can be moved from one system to another essentially without change.

We will not try to describe the entire I/O library here; we are more interested in showing the essentials of writing C programs that interact with their operating system environment.

7.1 Access to the Standard Library

Each source file that refers to a standard library function must contain the line

```
#include <stdio.h>
```

near the beginning. The file stdio.h defines certain macros and variables used by the I/O library. (The use of < and > instead of the double quotes implies a search for the file in a special place.) Furthermore, it may be necessary when loading the program to specify the library explicitly; for example, on Unix, the command to compile a program would be

```
cc [source files, etc.] -lS
```

where -lS indicates loading from the standard library.

7.2 "Standard Input" and "Standard Output" — getchar and putchar

The simplest input mechanism is to read a character at a time from the "standard input," which is generally the user's terminal, with **getchar**. **getchar( )** returns the next input character each time it is called. In most environments that support C, a file may be substituted for the terminal by using the '<' convention: if **prog** uses **getchar**, then the command line

```
prog <infile
```

causes **prog** to read **infile** instead of the terminal. **prog** itself knows nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

will provide the input for **prog** from the output of **otherprog**.

getchar returns the value **EOF** when it encounters end of file (or an error) on whatever input is being read. The standard library defines the symbolic constant **EOF** to be **−1**, but tests should be written in terms of **EOF**, not **−1**. so as to be independent of the specific value.

For output, **putchar(c)** puts the character **c** on the "standard output," which is also by default the terminal. The output can be captured on a file by using '>': if **prog** uses **putchar**,

```
prog >outfile
```

will write the output onto **outfile** instead of the terminal. On Unix or GCOS, a pipe can also be used:

```
prog | otherprog
```

puts the output of **prog** into the input of **otherprog**.

The function **printf**, which formats output in various ways, uses the same mechanism as **putchar** so output produced by **printf** also finds its way to the standard output, and calls to **putchar** and **printf** may be interleaved.

A surprising number of programs read only one input and write one output; for such programs I/O with **getchar**, **putchar**, and **printf** may be entirely adequate, and it is almost always enough to get started. This is particularly true given a pipe facility for connecting the output of one program to the input of the next. For example, here is a complete program that acts as a "filter" to strip out all ASCII control characters except newline and tab from its input. The program relies on the numerical properties of the ASCII character set — control characters are less than blank, or greater than or equal to octal 177.

```
main( )        /* strip out control characters */
{
        int c;

        while ((c = getchar( )) != EOF)
                if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
                        putchar(c);
}
```

If it is necessary to treat multiple files, you can use a program like the utility *cat* to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. (cat is presented later in this chapter.)

As an aside, in the standard I/O library the "functions" getchar and putchar are actually macros, and thus avoid the overhead of a function call per character.

7.3 Formatted Output — printf

The two routines printf for output and scanf for input permit translation to and from character representations of numerical quantities. They also allow generation or interpretation of formatted lines. We have used printf informally throughout the previous chapters; here is a complete description.

```
printf(control, arg1, arg2, ...)
```

printf converts, formats, and prints its arguments on the standard output under control of the control string. The control string contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf.

Each conversion specification is introduced by the character %. Following the % there may be:

- an optional minus sign '–' which specifies left adjustment of the converted argument in it field;
- an optional digit string specifying a minimum field width; if the converted argument has fewer characters than the field width it will be padded on the left (or right, if the left adjustment indicator has been given) to make up the field width; the padding character is blank normally and zero if the field width was specified with a leading zero (note that this does not imply an octal field width);
- an optional period '.' which serves to separate the field width from the next digit string;
- an optional digit string (the precision) which specifies the maximum number of characters to be printed from a string, or the number of digits to be printed to the right of the decimal point of a floating or double number.

- an optional length modifier *l* (letter ell) which indicates that the corresponding data item is a long rather than an int.
- a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are:

d The argument is converted to decimal notation.

o The argument is converted to octal notation (including a leading zero).

x The argument is converted to hexadecimal notation.

u The argument is converted to unsigned decimal notation.

c The argument is taken to be a single character.

s The argument is taken to be a string and characters from the string are printed until a null character is reached or until the number of characters indicated by the precision specification is exhausted.

e The argument is taken to be a float or double and converted to decimal notation of the form [–]m.nnnnnnE[–]xx where the length of the string of *n*'s is specified by the precision. The default precision is 6 and the maximum is 22.

f The argument is taken to be a float or double and converted to decimal notation of the form [–]mmm.nnnnn where the length of the string of *n*'s is specified by the precision. The default precision is 6 and the maximum is 22. Note that the precision does not determine the number of significant digits printed in f format.

g Use %e or %f, whichever is shorter; print no unnecessary zeros.

If no recognizable conversion character appears after the % that character is printed; thus % may be printed by %%.

7.4 Formatted Input - scanf

The function scanf is the input analog of printf, and provides many of the same conversion facilities.

```
scanf(control, arg1, arg2, ...)
```

scanf reads characters from the standard input, interprets them according to a format, and stores the results in its arguments. It expects a control argument, described below, and a set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

(1) Blanks, tabs or newlines, which are ignored.

(2) Ordinary characters (not %) which are expected to match the next non-space character of the input stream (space characters are blank, tab or newline).

(3) Conversion specifications, consisting of the character %, an optional assignment suppression character *, an optional numerical maximum field width, and a conversion character.

A conversion specification is used to direct the conversion of the next input field; the result is placed in the variable pointed to by the corresponding

argument, unless assignment suppression was indicated by the * character. An input field is defined as a string of non-space characters; it extends either to the next space character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. Pointers, rather than variable names, are required by the "call by value" semantics of the C language. The following conversion characters are legal:

% indicates that a single % character is expected in the input stream at this point; no assignment is done.

d indicates that a decimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

o indicates that an octal integer is expected in the input stream; the corresponding argument should be a integer pointer.

x indicates that a hexadecimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

s indicates that a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0 which will be added. The input field is terminated by a space character or a newline.

c indicates that a single character is expected; the corresponding argument should be a character pointer; the next input character is placed at the indicated spot. The normal skip over space characters is suppressed in this case; to read the next non-space character, try %1s.

e or f indicates that a floating point number is expected in the input stream; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for float's is an optional sign, a string of numbers possibly containing a decimal point, followed by an optional exponent field containing an E or e followed by a possibly signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o and x may be preceded by l (letter ell) to indicate that a pointer to long rather than int is expected. Similarly, the conversion characters e or f may be preceded by l to indicate that a pointer to double rather than float is in the argument list. The character h will function similarly in the future to indicate short data items.

For example, the call

```
int i;
float x;
char name[50];
scanf("%d %f %s", &i, &x, name);
```

with the input line

```
25   54.32E-1   thompson
```

will assign to i the value 25, x the value 5.432, and name will contain "thompson\0". Or,

```
int i;
float x;
char name[50];
scanf("%2d %f %*d %[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to i, 789.0 to x, skip "0123", and place the string "56\0" in name. The next call to any input routine will return a.

scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, -1 is returned; note that this is different from 0, which means that the next input character does not match what you called for in the control string.

7.5 In-memory Format Conversion

The functions scanf and printf have siblings called sscanf and sprintf which perform identical conversions, but operate on a string instead of a file. The general format is

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

sprintf formats the arguments in arg1, arg2, etc., according to control as before, but places the result in string instead of on the standard output. string of course had better be big enough to receive the result.

sscanf does the reverse conversions — it scans the string according to the format in control, and places the resulting values in arg1, arg2, etc. These arguments must be pointers.

7.6 File Access

The programs we have written so far have all read the standard input and written the standard output, which we have assumed are magically predefined for a program by the local operating system.

The next step in I/O is to write a program which accesses a file which is *not* already connected to the program. One simple example which clearly illustrates the need for such operations is **cat**, which concatenates a set of named files onto the standard output. **cat** is used as a general-purpose input collector for programs which do not have the capability of accessing files by name, and merely for listing things. For example, the command

```
cat foo gorp
```

will concatenate the contents of the files **foo** and **gorp** onto the standard output.

The question is how to arrange for the named files to be read — that is, how to connect the external names that a user thinks of to the statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function **fopen**. **fopen** takes the external name (like **foo**), does some housekeeping and negotiation with the operating system (details of which needn't concern us), and returns an internal name which must be used in subsequent reads or write of the file.

This internal name is actually a pointer to a structure which contains information about the file. We will call the pointer the *file pointer.* Users don't need to know the details of the structure that the file pointer refers to, for part of the standard I/O definitions obtained by **#include <stdio.h>** is a definition called **FILE**. The only declaration needed for a file pointer is exemplified by

```
FILE *fopen( ), *fp;
```

This says that **fp** is a pointer to a **FILE**, and **fopen** returns a pointer to a **FILE**.

The actual call to **fopen** is

```
fp = fopen("foo", "r");
```

The first argument of **fopen** is the name of the file as a character string. The second argument is the *mode,* which indicates whether one intends to read (**"r"**), write (**"w"**), or append (**"a"**) to the file.

If you open a file which does not exist for writing or appending, it is created (if possible). Opening a file that does not exist for reading is an error, and of course there may be other causes of error (like trying to read a file when you don't have permission). Any error causes **fopen** to return the null pointer value **NULL**.

The next thing needed is a way to read or write the file once it is open. There are several possibilities. The function **getc** returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in **c** the next character from the file referred to by **fp**. Like **getchar**, **getc** is a macro, not a function.

The macro **putc** is the inverse of **getc**:

```
putc(c, fp)
```

puts the character **c** on the file **fp** and returns **c**.

For formatted input or output on files, the functions **fprintf** and **fscanf** may be used. These are identical to **printf** and **scanf**, save that the first argument is a file pointer that specifies the file to be read or written.

With all of these preliminaries out of the way, we can actually write the program to concatenate files. The basic design that we have selected for **cat** is one that has been found convenient for many programs: if there are arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)     /* cat: concatenate files */
int argc;
char *argv[ ];
{
	FILE *fp, *fopen( );

	if (argc == 1)      /* no arguments; copy standard input */
		filecopy(stdin);
	else
		while (--argc > 0)
			if ((fp = fopen(*++argv, "r")) == NULL) {
				printf("cat: can't open %s\n", *argv);
				break;
			} else {
				filecopy(fp);
				fclose(fp);
			}
}
```

```
filecopy(fp)  /* copy fp to standard output */
FILE *fp;
{
	int c;

	while ((c = getc(fp)) != EOF)
		putc(c, stdout);
}
```

The file pointers **stdin** and **stdout** are pre-defined in the I/O library as the standard input and standard output; they may be used anywhere an object of type **FILE** can be. They are *not* variables, however, so don't try to assign to them.

getchar and **putchar** are defined in terms of **getc** and **putc** and **stdin** and **stdout** as follows:

```
#define	getchar	getc(stdin)
#define	putchar(c)	putc(c, stdout)
```

If you call any of these four "functions" with the wrong number of arguments, you will get some rather mysterious syntax error messages from the compiler.

The function **fclose** is the inverse of **fopen**; it breaks the connection between the file pointer and the external name that was established by **fopen**, freeing the file pointer for another file. Since most operating systems have some limit on the number of simultaneously open files that a program may have, it's a good idea to free things when they are no longer needed, as we did here in **cat**.

7.7 Error Handling — stderr and exit

The treatment of errors in **cat** is adequate, but not ideal. The trouble is that if one of the files can't be accessed for some reason, the diagnostic is at the end of the concatenated output. That is acceptable if that output is going to a terminal, but bad if it's going into another file.

To handle this situation in a better way, a second output file, called **stderr**, is assigned to a program in the same way that **stdin** and **stdout** are. If at all possible, output written on **stderr** appears on the user's terminal even when output for **stdout** is sent to a file.

Let us revise **cat** to handle errors better.

--

```
#include <stdio.h>

main(argc, argv)     /* concatenate files */
int argc;
char *argv[ ];
{
	FILE *fp, *fopen( );

	if (argc == 1)      /* no arguments; copy standard input */
		filecopy(stdin);
	else
		while (--argc > 0)
			if ((fp = fopen(*++argv, "r")) == NULL) {
				fprintf(stderr, "cat: can't open %s\n", *argv);
				exit(1);
			} else {
				filecopy(fp);
				fclose(fp);
			}
	exit(0);
}
```

The program signals errors two ways. The diagnostic output goes onto **stderr**, so it finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program also uses the standard library function **exit**, which terminates program execution when it is called. The argument of **exit** is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well, and various non-zero values signal abnormal situations.

7.8 The #include Processor

The next major example is a program which processes lines which contain **#include** statements. It is very easy for this program to handles nested **#include**'s by recursion, with a fresh level of recursion for each level of nesting.

Our version uses two new routines from the standard library. **fgets** is reminiscent of the **getline** function that we have used throughout the book. The call

```
fgets(line, MAXLINE, fp)
```

reads the next input line from file **fp** into the character array **line**; at most **MAXLINE** characters will be read. The resulting line is terminated with \0. Normally **fgets** returns **line**; on end of file it returns **NULL**.

fputs writes a line to a file:

```
fputs(line, fp)
```

and returns **line**.

We have also used **sscanf** to break out the first two strings of each input line rather than write a separate routine.

```
#include <stdio.h>
#define     MAXLINE     1000

main()      /* process #include */
{
      include(stdin);
      exit(0);
}

include(fp) /* include file fp */
FILE *fp;
{
      char s1[MAXLINE], s2[MAXLINE], line[MAXLINE];
      FILE *f, *fopen();
      int n;

      while (fgets(line, MAXLINE, fp) != NULL) {
            n = sscanf(line, "%s %s", s1, s2);
            if (n != 2 || strcmp(s1, "#include") != 0)
                  fputs(line, stdout);
            else if ((f = fopen(s2, "r")) == NULL) {
                  fprintf(stderr, "include: can't open %s\n", s2);
                  exit(1);
            } else {
                  include(f);
                  fclose(f);
            }
      }
}
```

By the way, there is nothing magic about the function **fgets**; it is just a C function. Here it is, copied directly from the I/O library:

```
#include     <stdio.h>

fgets(s, n, iop)
char *s;
register FILE *iop;
{
        register c;
        register char *cs;

        cs = s;
        while (--n>0 && (c = getc(iop))>=0) {
                *cs++ = c;
                if (c=='\n')
                        break;
        }
        if (c<0 && cs==s)
                return(NULL);
        *cs++ = '\0';
        return(s);
}
```

Exercise 7-1: Write a program to compare two files, printing the first line and character position where they differ. □

Exercise 7-2: Modify the pattern finding program of Chapter 5 to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found? □

Exercise 7-3: Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file. □